Jussi Pekkinen

Yksikkötestauksen mahdollisuus Finncomm Oy:ssä

Opinnäytetyö
Kevät 2011
Tekniikan yksikkö
Tietotekniikan koulutusohjelma

Seinäjoen ammattikorkeakoulu
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Koulutusohjelma: Tietotekniikan koulutusohjelma

Suuntautumisvaihtoehto: Ohjelmistotekniikan suuntautumisvaihtoehto

Tekijä: Pekkinen, Jussi

Työn nimi: Yksikkötestauksen mahdollisuus Finncomm Oy:ssä

Ohjaaja: Mäkelä, Petteri

Vuosi: 2011 Sivumäärä: 35 Liitteiden lukumäärä: 0

Opinnäytetyön tarkoituksena oli tutustua ohjelmistotestaukseen ja hieman tarkemmin yksikkötestaukseen. Tarkoitus oli myös tutkia, voisiko yksikkötestausta mahdollisesti toteuttaa Finncomm Oy:ssä.

Työn aikana tutustaan hieman teoreettisesti ohjelmistotestauksen teoriaan, ohjelmistovirheiden kustannuksiin ja ohjelmistotestauksen V-malliin. Hieman tarkemmin on käyty läpi, miten ohjelmistotestausta voidaan suorittaa ja millä tavoin. Koska opinnäytetyö liittyy enemmän yksikkötestaukseen, on yksikkötestausta käytä hieman tarkemmin läpi pienimuotoisen esimerkin avulla. Yksikkötestausta on suoritettu kahdella eri työkalulla.

Avainsanat: yksikkötestaus, ohjelmistotestaus

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Software Engineering

Author: Pekkinen, Jussi

Title of the thesis: Possibility of unit testing at Finncomm Oy

Supervisor: Mäkelä, Petteri

Year: 2011 Number of pages: 35 Number of appendices: 0

The purpose of the thesis was to get to know software testing and learn some more about unit testing. The purpose was also to study how unit testing could potentially be put into practise at Finncomm Oy.

In this thesis the theory behind software testing, costs of software bugs and the V-model of software testing was explored. How software testing would be accomplished was studied somewhat more precisely. Because this thesis is more about the unit testing it was covered in more detail and with an example. Two unit testing tools were used in this thesis.

Keywords: unit testing, software testing

SISÄLTÖ

Kuvio- ja taulukkoluettelo

Käytetyt termit ja lyhenteet

.NET Framework Microsoftin kehittämä luokkakirjasto, joka sisältää ohjelmistokomponentteja (Drayton 2002).

C# C# on Microsoftin kehittämä oliopohjainen ohjelmointikieli (Drayton 2002).

Luokka Olio-ohjelmoinnissa luokka kuvaa olion tyyppiä. Luokka voi sisältää ominaisuuksia, attribuutteja ja metodeja (Räsänen 2007).

Metodi Metodi käsittelee olion sisältämää tietoa olio-ohjelmoinnissa. Metodi voi palauttaa arvon tai sitten vain suorittaa tietyn toiminnan (Räsänen 2007).

1 JOHDANTO

1.1 Työn tausta

Testauksen merkitys ohjelmistokehityksessä on merkittävä. Jos ohjelma julkaistaan ilman testausta, niin voi olla varma, että ohjelmistoyritys ei pysy kilpailussa mukana. Ohjelmistotestauksen laiminlyöminen voi korottaa negatiivisten yhteydenottojen määrää tai mahdollisesti korvausvaatimuksia ohjelman käyttäjiltä. Vaikka ohjelmaa ohjelmoitaessa tulee osittain suoritettua testejä, se ei riitä. Testaukseen on varattava tarpeeksi aikaa, jotta testaus voidaan suorittaa mahdollisimman hyvin.

Ohjelmoijalle yksikkötestaus on helpoin tapa, koska yleensä testattavat kohteet on helppo määritellä samalla kun kirjoittaa ja suunnittelee ohjelmakoodia. Pelkästään testaaminen yksikkötestauksen avulla voi ohjelman myöhemmässä vaiheessa säästää monilta turhilta virheiltä, jotka oli pitänyt huomata jo ohjelman ohjelmointivaiheessa. Varsinkin isommissa ohjelmissa, joissa on kymmeniätuhansia koodirivejä, on virheiden etsiminen myöhemmässä vaiheessa todella hankalaa ja turhauttavaa. Vaikka jokaista virhettä ei yksikkötestauksella löydäkään, on silti siitä suuri apu.

Opinnäytetyössä keskitytään kahteen yksikkötestaustyökaluun, jotka ovat molemmat tehty C#-ohjelmointikielellä ja myös esimerkit ovat tehty C#-kielellä. C#-ohjelmointikieli valittiin siksi, että se on ainut ohjelmointikieli, joka on käytössä yrityksessä.

1.2 Työn tavoite

Tässä opinnäytetyössä on tarkoitus tutkia yksikkötestauksen mahdollisuutta kahdella eri ohjelmalla ja tutkia niiden mahdollista käyttöä tulevissa ohjelmistoprojekteissa yrityksessä. Tarkoitus on myös tutkia, miten paljon yksikkötestauksen suunnitteleminen ja toteuttaminen vie aikaa ja resursseja. Lisäksi tarkoituksena olisi

syventää ja oppia hieman enemmän ohjelmistotestauksesta ja tarkemmin yksikkötestauksesta.

1.3 Työn rakenne

Toisessa luvussa käydään lävitse ohjelmistotestausta yleisesti ja kerrotaan hieman ohjelmistotestauksen V-mallista.

Kolmannessa luvussa kerrotaan ohjelmistotestauksen strategioita, joihin kuuluvat muun muassa musta-/lasi- ja harmaalaatikkotestaus. Lisäksi kerrotaan ohjelmistotestauksen testaustasoista ja miten ne liittyvät ohjelmistotestauksen strategioihin. Ohjelmistotestauksen testaustasoja käydään tarkemmin lävitse luvussa 4.

Luvussa 5. kerrotaan Visual Studion ja NUnitin yksikkötestaustyökaluista. Nämä työkalut valittiin siksi, että NUnit on yksi tunnetuimmista yksikkötestauksen työkaluista ja Visual Studio sen vuoksi, että yrityksessä siirrytään käyttämään uusimmissa projekteissa 2010-versiota Visual Studiosta. Samassa luvussa vertaillaan näitä kahta työkalua esimerkkimallin avulla. Kahdessa viimeisessä luvussa käsitellään työn tuloksia ja viimeisessä luvussa on yhteenveto tästä opinnäytetyöstä.

1.4 Finncomm Oy

Finncomm Oy on vuonna 1993 perustettu lentoyhtiö. Finncomm Oy:ssä työskentelee yli 300 henkilöä. Yhtiön pääkonttori sijaitsee Seinäjoella ja laivaston kotikenttä sekä operatiivinen keskus on Helsingissä. Finncomm Oy tekee yhteistyötä toisen suomalaisen lentoyhtiön, Finnairin kanssa ja hoitaa syöttöliikennettä Finnairille. Yhtiöllä on tällä hetkellä käytössä 12 ATR- ja kaksi Embraer-lentokonetta. Finncomm lentää pääosin kotimaisilla kentillä, mutta myös muutamaan ulkomaiseen kohteeseen on lentoja. (Holkko 2011.)

2 OHJELMISTOTESTAUS

Tässä luvussa kerrotaan ohjelmistotestauksesta ja ohjelmistotestauksen V-mallista.

2.1 Ohjelmistotestauksesta yleisesti

Ohjelmistotestauksella varmistetaan ja validoidaan ohjelma, jotta se vastaa määrittelyn vaatimuksia ja toimii niin kuin se on suunniteltu toimimaan. Välttämättä jokaista virhettä ohjelmistotestauksella ei saada korjattua, mutta selkeät ja toistuvat virheet ohjelmakoodissa on mahdollista korjata. Ohjelmistotestauksella on suuri merkitys ohjelmistoprosessissa ja huonolla testaamisella voi olla dramaattinen vaikutus yrityksen imagoon. (Bentley 2005.)

Myersin (2004, 8-12) mielestä myös ihmisen asenteella ja psykologialla on vaikutusta ohjelmistotestaamiseen. Yksi pääsyistä huonoon ohjelmistotestaukseen on muun muassa ohjelmoijan asenne testaamiseen. Ohjelmoija voi ajatella, että hänen työnsä jälki ei ole riittävä ja testaamisella vain osoitetaan ohjelmoijan tekemät virheet. Asia ei ole kuitenkaan ole näin. Olisi tärkeää, että testaaja ei testaa ohjelmaa tai laitetta vain näyttääkseen, että se toimii. Ennen testausta testaajan tulisi ajatella, että ohjelma sisältää virheitä ja vasta sen jälkeen aloittaa kunnon testaaminen ja yrittää löytää mahdollisimman paljon virheitä. (Myers 2004, 8-12.)

Myers (2004, 11) tiivistää ohjelmistotestaamisen hyvin yhteen lauseeseen: Testaus on ohjelman suorittamisen prosessi, jossa aikomuksena on löytää virheitä. (Myers 2004, 11.)

Ohjelmistotestauksen todellisen merkityksen ymmärtämisellä voi olla huomattavan suuri ero henkilökohtaiseen panostukseen. Jos testaajan tavoite on näyttää ohjelman olevan virheetön, niin ihminen alitajuisesti pyrkii tähän. Testaaja suunnittelee testattavaa dataa sillä tavoin, että virheitä ei mahdollisesti tapahdu. Toisaalta, jos päämääränä on virheetön ohjelma, niin testattavalla datalla voidaankin saada hyvin todennäköisesti ilmentymään virheitä. (Myers 2004, 11.)

Myers (2004, 11) esittelee myös pari oletuslausetta liittyen ohjelmistotestaukseen. **Testaus on osoittamisen prosessi, että virheitä ei näy**. Ilmaisu on mahdoton jokaiselle ohjelmalle. Psykologisilla testeillä on osoitettu, että ihmiset suoriutuvat huonosti, jos heidät määrätään suorittamaan sellaista tehtävää, jonka he tietävät olevan mahdoton. **Testaus on osoittamisen prosessi, että ohjelma tekee sen mitä sen pitääkin tehdä.** Vaikka ohjelma tekisikin sen mitä sen pitää tehdä, voi se silti sisältää virheitä. (Myers 2004, 11.)

Hyvänä esimerkkinä ohjelmistotestauksesta voisi mainita lentokoneiden ohjelmistot. Ilman todella tarkkaa ja tiukkaa ohjelmistotestausta eivät lentokoneiden valmistajat laittaisi niin paljon tietotekniikka lentokoneisiin ja antaisi siirtää ihmisen vastuuta enemmän tietokoneille.

Nykyään eivät kaikki ohjelmistotestaajat lue itse ohjelmakoodia, mutta on toki hyväksyttävää, että testaaja tutkii ohjelmakoodia testausprosessin aikana. Riippuen muun muassa ohjelmasta, ohjelmakoodin laajuudesta, ohjelmistokehittäjien lukumäärästä tai ohjelman kehittämiseen annetusta ajasta, testaaja voi jättää väliin ohjelmakoodin tutkimisen ja keskittyä enemmän mustalaatikkotestaukseen. (Myers 2004, 21.) Mustalaatikkotestausta käydään tarkemmin lävitse luvussa 3.

Bentley (2005) esittelee kolme ohjelmistotestausprosessiin kuuluvaa käsitettä, jotka ovat varmistus, validointi ja virheiden etsiminen. Varmistuksella tarkoitetaan sitä, että ohjelma vastaa ohjelmasta tehtyä teknistä määrittelyä. Teknisessä määrittelyssä kuvataan ohjelman tai ohjelmiston arkkitehtuurin mahdollisimman tarkasti. (Bentley 2005.)

Validaatiolla tarkoitetaan sitä, että ohjelma vastaa ohjelmaa käyttävän yrityksen vaatimuksia. Mitä tahansa ohjelmaa kehitetäänkään, ohjelman pitäisi toiminnallisesti tehdä sen, mitä sen pitäisikin tehdä ja sen pitäisi tyydyttää kaikki toiminnalliset vaatimukset, jotka käyttäjät ovat määritelleet. (Parekh 2005.)

Tässä tapauksessa virhe määritellään siten, että jos odotettu ja oikea lopputulos eivät täsmää, niin on tapahtunut virhe. Tapahtunut virhe voidaan jäljittää joko virheeseen teknisessä määrittelyssä, suunnittelussa tai itse ohjelman ohjelmointivaiheessa. (Bentley 2005.)

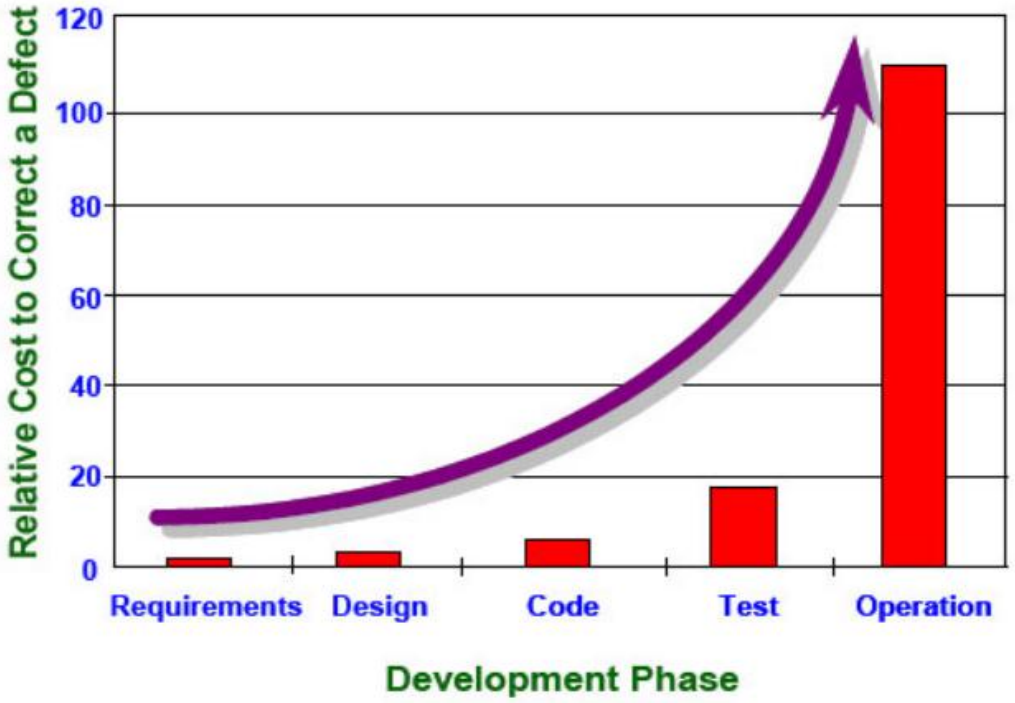
Myers (2004, 16-17) on esittänyt kymmenen hyvää ohjelmistotestauksen periaatetta.

1. Testitapauksen pakollinen osa on määritelmä oletetusta tuloksesta.
2. Ohjelmoijan olisi vältettävä testausta ohjelmasta, jonka hän on itse tehnyt.
3. Ohjelmistoyrityksen olisi vältettävä testausta ohjelmista, jota yritys on tuottanut.
4. Käy tarkasti läpi jokaisen testin tulokset.
5. Testitapaukset on suunniteltava siten, että ohjelmalle syötettävät arvot ovat vääriä ja virheellisiä ja myös niin, että syötettävät arvot ovat oikeita ja odotettuja.
6. Ohjelman tutkiminen nähdäkseen sen, että ohjelma ei tee niin kuin sen pitäisi tehdä, ei riitä vielä. Vielä pitäisi kokeilla, että ohjelma ei tee niin kuin sen ei pitäisi tehdä.
7. Vältä ”kertakäyttöisiä” testitapauksia.
8. Älä testaa siinä mielentilassa, että ohjelmasta ei löydy virheitä.
9. Mahdollisten virheiden todennäköisyys ohjelman osassa kasvaa suhteessa siihen, montako virhettä on jo löytynyt siitä.
10. Testaus on erittäin luovaa ja haastava tehtävä. (Myers 2004, 16-17.)

Kokemuksena voisi sanoa, että varsinkin ohjelman loppuvaiheessa ei itse saisi testata ohjelmaa. Parempi olisi antaa sellaisen henkilön testata, joka ei ole ollut mukana samassa projektissa. Ihmisen aivot käyttäytyvät mielenkiintoisesti, koska jossain testauksen vaiheessa tulee ”sokeaksi” omalle ohjelmalleen. Ohjelmaa tulee testattua alitajuisesti. Vaikka yrittää etsiä jokaista mahdollista virhettä, niin alitajunta estää sen. Oma alitajunta tietää miten ohjelmassa pitäisi siirtyä tietyn toiminnan jälkeen ja minkälaisia arvoja pitää syöttää ohjelmaan, jotta oikeanlainen tulos syntyisi. Tämän vuoksi ohjelmaa, jota on ollut itse kehittämässä, pitäisi aina testata joku muu henkilö kuin ohjelmoija.

2.2 Ohjelmistovirheiden kustannukset

Kalliina esimerkkinä voitaisiin mainita vuonna 1996 Ariane 5 -raketin räjähtäminen muutamien kymmenien sekuntien jälkeen laukaisusta. Raketin räjähtäminen johtui virheestä ohjelmakoodissa. (Hatton 1999.) Kunnolla ohjelmistotestauksella olisi mahdollisesti voitu välttää tämä kallis virhe. Ohjelmistotestauksen laiminlyömisellä voi olla todella suuri merkitys myös rahassa.

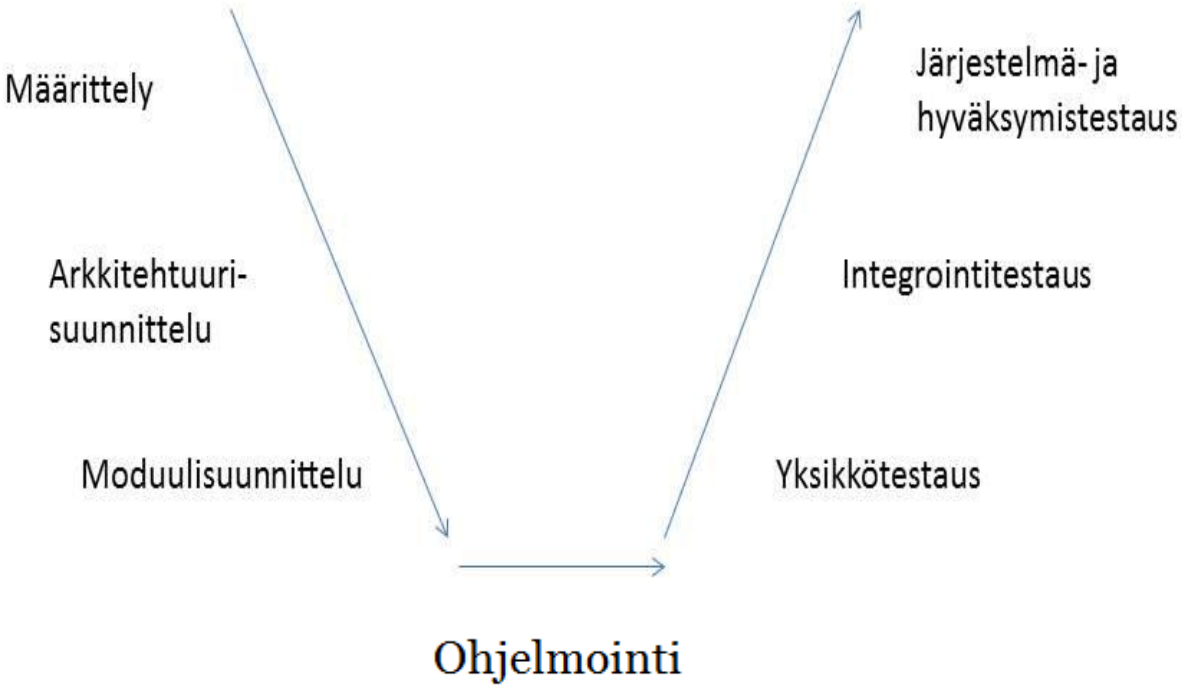


Kuvio 1. Kustannusarviokaavio (LKP Consulting Group 2006).

Kuviosta 1 nähdään ohjelmistovirheiden suhteellisen kustannusarvion (Relative Cost to Correct a Defect) suhteessa ohjelmiston kehitysvaiheisiin (Developement Phase). Mitä lähemmäksi mennään käyttöönottovaihetta, sitä kalliimmaksi virheiden korjaus tulee.

2.3 Ohjelmistotestauksen V-malli

V-malli on ohjelmistotuotantoprosessi, jonka voidaan olettaa olevan vesiputousmallin lisäjatke. V-malli esiteltiin 1980-luvun loppupuolella Paul E. Rookin toimesta (Meerts 2010). V-mallin kuvaa ohjelmistokehityksen ja ohjelmistotestauksen välistä suhdetta. Mallin tarkoituksena on kehittää ohjelmistokehityksen tehokkuutta ja



Kuva 1. Ohjelmistotestauksen V-malli

Jokainen kehitysvaihe varmistetaan ennen kuin siirrytään seuraavaan vaiheeseen mallissa. V-mallissa testaus alkaa heti määrittelyn jälkeen ja jatkuu ohjelman valmistumiseen asti. Jos testauksen avulla huomataan mahdollisia virheitä prosessin alkuvaiheessa, niin mahdollisten virheiden ilmentyminen ohjelmistoprosessin myöhemmissä vaiheissa voi vähentyä huomattavasti. Jokaiselle kehitysvaiheelle, eli vaiheille jotka ovat V-mallin vasemmalla puolella, on vastaava testaussuunni-

telman V-mallin oikealla puolella. Kun kehitysvaihetta käydään lävitse, samalla suunnitellaan testaussuunnitelmaa tälle kehitysvaiheelle myöhempää testausta varten. (Ghahrai 2008.)

V-mallin hyödyt ovat monipuoliset. Jokaisella kehitysvaiheella on oma testaussuunnitelma. Käyttämällä V-mallia on suuri mahdollisuus onnistua hyvän testaussuunnitelman vuoksi. V-malli vie vähemmän aikaa verrattuna vesiputousmalliin ja toimii hyvin pienemmissä ohjelmistoprojekteissa, jossa vaatimukset ovat helposti ymmärrettävissä. (Ghahrai 2008.)

V-mallissakin on myös huonoja puolia. V-malli on joustamaton, niin kuin vesiputousmallikin. Ohjelmaa kehitetään toteutusvaiheessa, joten aikaisia prototyyppejä ei välttämättä synny ja V-malli ei tarjoa oikeaa ja selkeää opasta mahdollisen virheiden sattuessa testausvaiheessa. (Ghahrai 2008.)

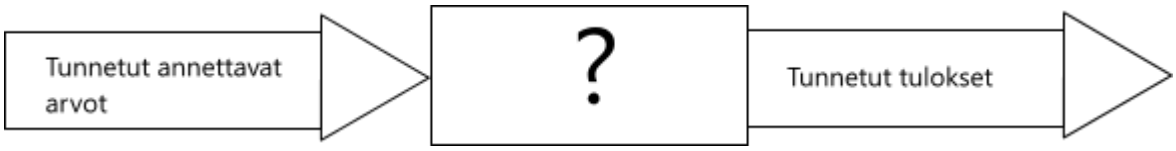
3 OHJELMISTOTESTAUKSEN STRATEGIAT

Yleisimpiin ohjelmistotestauksen strategioihin kuuluvat mustalaatikkotestaus, lasilaatikkotestaus ja harmaalaatikkotestaus.

3.1 Mustalaatikkotestaus

Kuten Myers (2004, 13) kertoo, kuvitellaan ohjelma mustana laatikkona. Tavoitteena ei ole huolehtia ohjelman rakenteesta ja siitä kuinka ohjelman ”sisällä” käyttäydytään. Tavoitteena on keskittyä sellaisiin kohtiin, joissa ohjelma ei käyttäydy niin kuin sen pitäisi käyttäytyä vaatimusten mukaan. Testidata suunnitellaan pelkästään ohjelman vaatimuksista/teknisistä tiedoista, eikä käytetä hyväkseen tietoa ohjelman sisäisestä käyttäytymisestä. (Myers 2004, 13.)

Kuvassa 2 testaaja syöttää ohjelmalle tietoa ja tarkastelee antaako ohjelma oikeat tulokset syötettyjen tietojen perusteella. Testaajan ei tarvitse tietää kuinka se tieto käsitellään vaan, että ohjelman antamat tulokset ovat oikein.



Kuva 2. Mustalaatikkotestaus.

Mustalaatikkotestauksen olisi parempi suorittaa joku muu henkilö, kuin ohjelmistosuunnittelija, joka on ollut tekemässä ohjelmaa. Ongelma on se, että ohjelmistosuunnittelija tietää liian hyvin ohjelman sisäisen toiminnan, joka ei ole eduksi mustalaatikkotestauksessa. (Parekh 2007.)

Mustalaatikkotestaus on tehokas isommissa ohjelmissa tai järjestelmissä. Testaajan ei välttämättä tarvitse olla asiantuntija, eikä tieto ohjelman tai järjestelmän toiminnasta ole pakollista. Testaus tapahtuu ohjelman tai järjestelmän käyttäjän näkökulmasta, ja testidataa voi alkaa suunnitella heti, kun toiminnallinen määrittely on tehty. (Koundinya 2010.)

Mustalaatikkotestauksen huonoihin puoliin kuuluvat muun muassa testitapauksien suunnittelun vaikeus, koska tietoa ohjelman tai järjestelmän sisäisistä toiminnoista ei ole. On suuri todennäköisyys tehdä samoja testejä, mitä ohjelmoija on jo tehnyt, ja on vaikea tunnistaa jokaista mahdollista ohjelmalle tai järjestelmälle syötettävää tietoa ajan puitteissa. Testitapauksien suunnittelu voi olla vaikeaa ja hidasta. (Koundinya 2010.)

Mustalaatikkotestaukseen kuuluu muun muassa järjestelmätestaus.

3.2 Lasilaatikkotestaus

Lasilaatikkotestauksessa käytetään hyväksi ohjelman sisäistä toimintaa. Testausdata suunnitellaan ohjelman logiikan perusteella ja yleensä päinvastoin mitä ohjelman teknisissä tiedoissa sanotaan. Ohjelman testaajan on tiedettävä enemmän itse ohjelman sisällöstä ja tarvittaessa löydettävä oikea paikka ohjelmakoodista, jossa virhe esiintyy. Lasilaatikkotestauksen olisi hyvä suorittaa esimerkiksi ohjelman tai järjestelmän ohjelmoija. (Myers 2004, 14.)

Esimerkkinä voidaan käyttää kuvaa 2, mutta nyt testaajan on tiedettävä ohjelman käyttäytymisestä oikean ja virheellisen testidatan perusteella. Itse ohjelma ja sen toiminnot sijaitsevat kuvan 2 kysymysmerkin kohdalla.

Lasilaatikkotestauksessa on muutamia etuja. Koska testaajan on tiedettävä ohjelman sisäisestä logiikasta, on testidatan suunnitteleminen ja tekeminen helpompaa. Lasilaatikkotestaus auttaa myös optimoimaan ohjelmakoodia. Haittana on se, että on melkein mahdotonta katsoa jokaista koodiriviä erikseen löytääkseen virheitä, jotka voivat aiheuttaa ongelmia myöhemmässä vaiheessa. (Parekh 2006.)

Lasilaatikkotestausta voidaan käyttää yksikkötestaukseen ja integrointitestaukseen.

3.3 Harmaalaatikkotestaus

Harmaalaatikkotestaus sijaitsee lasi- ja mustalaatikkotestauksen välissä. Harmaalaatikkotestaus yhdistää molempien testauksien osia. Harmaalaatikkotestaus ei ole pelkästään mustalaatikkotestausta, koska testaaja tietää osia järjestelmän tai ohjelman sisäisistä toiminnoista. Harmaalaatikkotestaus ei myös ole pelkästään lasilaatikkotestausta, koska testaaja tietää kuinka ohjelman pitäisi toimia käyttäjän näkökulmasta. Harmaalaatikkotestauksessa testaaja tekee osan testauksista ohjelman tai järjestelmän sisäisiin tekniikoihin ja osan ulkoisiin, eli testaa muun muassa käyttäjän näkökulmasta. Harmaalaatikkotestauksen hyötyihin ja haittoihin kuuluvat lasi- ja mustalaatikkotestauksien hyödyt ja haitat. (Davis 2000.)

4 OHJELMISTOTESTAUKSEN TESTAUSTASOT

Tässä luvussa esitellään muutaman tärkeimmän ohjelmistotestauksen testaustasot: yksikkötestauksen, integrointitestauksen, järjestelmätestauksen ja hyväksymistestauksen.

4.1 Yksikkötestaus

Yksikkötestauksen päämääränä on ottaa pieni osa ohjelman koodista, esimerkiksi luokka tai funktio, eristää se muusta ohjelmasta ja katsoa toimiiko se toivotulla tavalla. Jokainen tarvittava koodin osa testataan erikseen ennen kuin ne liitetään isommaksi moduuliksi, jota testataan sitten muiden moduulien kanssa. Yksikkötestauksen tarkoituksena on löytää ja korjata mahdolliset virheet aikaisessa vaiheessa. Virheiden etsiminen isommassa moduulissa voi olla työlästä, joten suositeltavaa olisi pilkkoa ohjelmakoodi pienempiin paloihin, testata ne, yhdistää ohjelmakoodi isommaksi moduuliksi ja testata moduuli. Tämä helpottaa huomattavasti virheiden etsintää. Yksikkötestauksessa luokan tai funktion ohjelmoija suorittaa yksikkötestauksen samalla kun kirjoittaa ohjelmakoodia. (MSDN 2003a.)

Myers (2004, 70) mainitsee muutamia asioita, jotka antavat motivaatiota yksikkötestauksen suorittamiseen ja toteuttamiseen. Ensimmäisenä, yksikkötestaus on tapa hoitaa testauksen yhdistyneitä osia, koska huomio keskittyy pienempiin osiin ohjelmakoodia. Toisena asiana on, että yksikkötestaus helpottaa ohjelmakoodin virheiden etsintää, koska virheen löytyessä sen paikka löytyy tietyssä moduulissa. Kolmantena on, että moduulien testausta voidaan suorittaa rinnakkain, eikä vain yhtä moduulia kerrallaan. (Myers 2004, 70.)

Yksikkötestausta käsitellään myöhemmässä kappaleessa hieman enemmän, kahden eri välineen avulla ja hieman enemmän käytännössä.

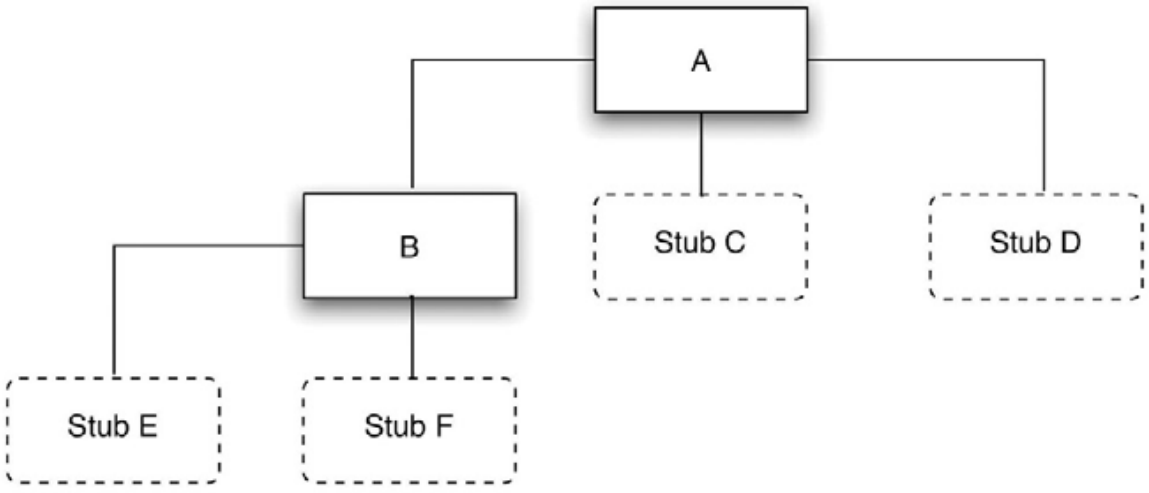
4.2 Integrointitestaus

Integrointitestauksessa yhdistellään jo testattuja moduuleja ja niitä testataan, kunnes niistä tulee isompi osa ohjelmaa. Vaikka moduulit toimisivat oikein, voi silti tulla ongelmia, kun moduulien välisiä rajapintoja ja kommunikaatiota testataan. Eli integrointitestauksessa testataan enimmäkseen moduulien välisiä rajapintoja ja moduulien välistä kommunikointia. (MSDN 2003b.)

Integrointitestauksella saadaan huomattua mahdolliset virheet, joita voi tapahtua kun moduulit yhdistetään isommiksi moduuleiksi. Integrointitestaus vähentää testausta, mutta virheiden löytäminen moduulien sisältä voi olla vaikeampaa. (MSDN 2003b.)

Integrointitestaus voidaan tehdä kahdella eri tavalla: ylhäältä alaspäin ja alhaalta ylöspäin.

4.2.1 Ylhäältä alaspäin -testaus



Kuva 3. Kuva ohjelman moduuleista (Myers 2004, 83).

Kuvasta 3 nähdään, ylhäältä alaspäin -integrointitestauksessa aloitetaan testaamalla ylintä moduulia ensin ja siitä siirtymällä alaspäin alempiin moduuleihin. Siirryttäessä alempiin moduuleihin ei ole oikeaa tapaa siirtyä. Sääntönä on, että tes-

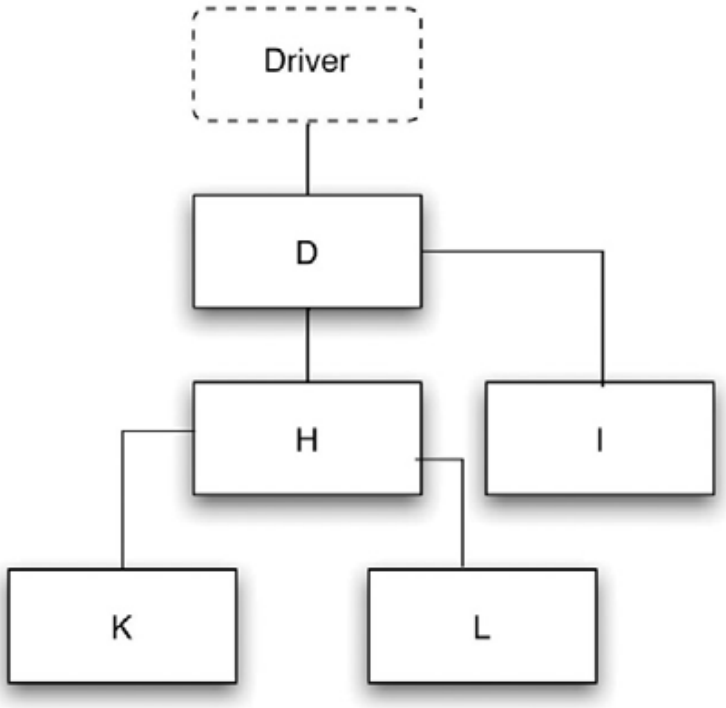
tattavaan moduuliin siirryttäessä on sen moduulin, joka kutsuu testattavaa moduulia, olla jo testattuna. (Myers 2004, 82-83.)

Esimerkiksi, kun kuvassa 3 olevan moduuli A on testattu, oikea moduuli korvaa yhden tynkämoduulin (Stub). Eli kuvan 3 perusteella moduuli A testauksen jälkeen on korvattu tynkä-B oikealla moduulilla.

Ylhäältä alaspäin -testauksessa on muutama ongelma. Kun ohjelman ”pienempiä” osia suunnitellaan, joissain tilanteissa voidaan huomata, että on tarve muuttaa tai parantaa ”ylempien” osien toimintaa. Jos ”ylemmät” osat on jo testattu ja ohjelmoitu, niin mahdolliset parantavat muutokset voidaan joutua hylkäämään. (Myers 2004, 85.)

Toisena ongelmana on, että moduulia ei välttämättä testata kunnolla ennen siirtymistä seuraavaan moduuliin. Tämä johtuu siitä, että testattavaa dataa ei välttämättä saada kunnolla alempiin moduuleihin, joita ei ole vielä mahdollisesti kunnolla tehty ja koska ”ylemmät” moduulit tarjoavat tarvittavaa tietoa ”alemmille” moduuleille. (Myers 2004, 85.)

4.2.2 Alhaalta ylöspäin -testaus



Kuva 4. Toinen kuva ohjelman moduuleista (Myers 2004, 83).

Alhaalta ylöspäin -testaus on enimmäkseen päinvastainen kuin ylhäältä alaspäin -testaus. Ylhäältä alaspäin -testauksen edut eivät ole enää etuja alhaalta ylöspäin -testauksessa ja ylhäältä alaspäin -testauksen huonot puolet ovat alhaalta ylöspäin -testauksen hyvät puolet. (Myers 2004, 86.)

Alhaalta ylöspäin -testaus alkaa ohjelman alemmista moduuleista, eli niistä moduuleista jotka eivät kutsu muita vaan niitä kutsutaan. Näiden moduulien testauksien jälkeen ei tässäkään ole parhainta tapaa siirtyä ylempiin moduuleihin. Ainoana sääntönä pidetään, että testattavan moduulin alimmat moduulit, eli moduulit joita testattava moduuli kutsuu, on testattu ennemmin. Alhaalta ylöspäin -testauksen ongelma on, että ohjelma ei ole olemassa ennen kuin viimeisin, eli ylimmäisin moduuli on lisätty. (Myers 2004, 87.)

Esimerkiksi voitaisiin ensin testata moduulit K ja L ja siirtyä koko ajan ylöspäin kohti moduulia A (kuvassa 4 Driver).

Kaikki ongelmat testiskenaarioiden luomisessa ylhäältä alaspäin -testauksessa eivät päde alhaalta ylöspäin -testaukseen. Kun lähdetään liikkumaan alemmasta moduulista ylöspäin, niin ei tarvitse huolehtia kuinka ylemmät moduulit vaikuttavat testaukseen. (Myers 2004, 87.)

4.3 Järjestelmätestaus

Järjestelmätestauksella ei tarkoiteta ohjelman tai järjestelmän funktioiden ja toimintojen testausta. Järjestelmätestauksella on tietty tarkoitus, eli vertailla ohjelmaa tai järjestelmää sen alkuperäisiin tavoitteisiin, jotka ovat määritelty määrittelydokumentissa. (Myers 2004, 96.)

Järjestelmätestauksessa siirrytään ensimmäistä kertaan monen käyttäjän testaukseen, eli siirrytään lähemmäs ohjelman tai järjestelmän oikeaa käyttöä. Tarkoituksena on testata ohjelma tai järjestelmä monella eri käyttäjällä samanaikaisesti ja rasittaa ohjelmaa tai järjestelmää mahdollisimman paljon. Järjestelmätestaus näkee testattavan ohjelman tai järjestelmän sellaisen käyttäjän silmin, joka ensimmäistä kertaa kokeilee sitä. Testaajien on myös varmistettava, kuinka ohjelma tai järjestelmä toimii esimerkiksi sähkökatkoksen jälkeen tai kuinka ohjelma toimii yrityksen tietokoneessa. Eli onko tietokoneiden laitteisto yhteensopiva ohjelman kanssa. (Loveland 2005, 32-33.)

Järjestelmätestaukseen kuuluu myös ohjelmiston tai laitteiston turvallisuus- ja käytettävyystestaus. Turvallisuustestauksen tavoitteena on varmistaa, että esimerkiksi ohjelman kautta ei voida hyökätä käyttäjän tietokoneeseen ja sitä kautta mahdollisesti yrityksen yksityisiin tietoihin. Varsinkin internetsovellukset on tarkistettava todella huolellisesti, etenkin jos sivulla voi vierailla kuka tahansa henkilö. (Myers 2004, 99-101.)

Käytettävyydellä tarkoitetaan sitä, että kuinka ohjelman tai laitteen oikea käyttäjä osaa käyttää ohjelmaa. Kuinka esimerkiksi käyttäjä näkee ohjelman valikot tai dialogit. Ovatko ne yksinkertaisia ja selkeitä vai tapahtuvatko toiminnot monen mutkan kautta. Ohjelman tai järjestelmän käytettävyydellä on todella suuri merkitys.

Muun muassa ohjelman käyttöliittymä voi vaikuttaa negatiivisesti ohjelman tekijöiden imagoon. (Myers 2004, 99-101.)

Osana järjestelmätestausta on myös testi, jossa ohjelmalle tai laitteelle syötetään suuri määrä dataa. Esimerkiksi ohjelman kääntäjälle annetaan suurikokoinen ohjelmakoodi suoritettavaksi. Järjestelmätestauksen tarkoituksena on saada ohjelma tai laite rikkoutumaan, jos alkuperäisessä määrittelyssä on mainittu jotain tähän testaukseen liittyen. Jokaiselle ohjelmalle tai laitteelle olisi suoritettava tämänlaisia testejä, vaikka sitä ei olisi määriteltykään. (Myers 2004, 98.)

4.4 Hyväksymistestaus

Hyväksymistestauksessa henkilö, joka on saanut ohjelman testattavaksi suorittaa testejä vertailemalle ohjelman toimintoja alkuperäiseen dokumenttiin. Esimerkiksi yrityksen työntekijä, joka vastaanottaa ohjelman ohjelmantekijältä, testaa ohjelmaa miten se suoriutuu alkuperäisen määrittelydokumentin mukaan. Paras tapa suorittaa hyväksymistestausta, olisi suunnitella testitapaukset siten, että yrittää näyttää toteen ohjelman vääränlaisen toiminnan verrattuna määrittelydokumenttiin. Hyväksymistestaus suoritetaan yleensä joko osana järjestelmätestausta tai vasta järjestelmätestauksen jälkeen. (Myers 2004, 104.)

5 YKSIKKÖTESTAUSTA KÄYTÄNNÖSSÄ

5.1 Testattavat välineet

Testattaviksi välineiksi valittiin avoimen lähdekoodin ohjelma NUnit ja Visual Studio 2010:n mukana tulevat yksikkötestaustyökalut. NUnit on yksi suosituimmista yksikkötestaus työkaluista Microsoftin .NET Frameworkille ja Visual Studio 2010:n yksikkötestaus onnistuu taasen helposti, koska Visual Studion käyttäminen on tuttua ja helppoa yrityksessämme.

Näiden kahden työkalun vertailu on hieman vaikeaa teoriassa, joten lukijan kannalta on helpompi kertoa asia oikean esimerkin kautta. Tarkoituksena on luoda yksinkertainen matemaattinen luokka, antaa luokalle metodeja ja testata näitä. Tätä luokkaa testataan aluksi Visual Studion yksikkötestaustyökalulla ja sen jälkeen NUnitin avulla.

Taulukko 1. Attribuuttien vertailu

Visual Studio 2010	NUnit	Tarkoitus
[TestMethod]	[Test]	Määrittää yksittäisen yksikkötestausmetodin
[TestClass]	[TestFixture]	Määrittää joukon yksikkötestausmetodeja

NUnitin ja Visual Studio 2010:n yksikkötestaukseen liittyvissä attribuuteissa ei ole kovin paljoa eroa (Taulukko 1). Ainoastaan nimet hieman poikkeavat toisistaan, mutta attribuutit tekevät kuitenkin saman asian. Testeissä käytettävät Assert-funktiot suorittavat testin ja vertailun lisäksi kertovat ohjelmalla, onko testi onnistunut.

5.2 Testattava luokka

```
namespace ClassLibrary2
{
    public class MatikkaLuokka
    {
        public int PlussaaNumerot(int a, int b)
        {
            return a + b;
        }


        public int MiinustaNumerot(int a, int b)
        {
            return a - b;
        }


        public int KerroNumerot(int a, int b)
        {
            return a * b;
        }

        public int JaaNumerot(int a, int b)
        {
            return a / b;
        }
    }
}
```

Kuva 5. Käytettävä matemaattinen luokka.

Aluksi on luotu yksinkertainen luokka, joka sisältää matemaattisia metodeja. Tätä luokkaa käytetään hyväksi molemmissa työkaluissa. Jokaiselle metodille annetaan kaksi lukua. PlussaaNumerot-metodi palauttaa niiden summan, MiinustaNumerot-metodi palauttaa lukujen erotuksen, KerroNumerot-metodi palauttaa lukujen tulon ja JaaNumerot-metodi palauttaa lukujen jakolaskun tuloksen. Metodit ovat tehty mahdollisimman yksinkertaisiksi, jotta lukijan on helpompi ymmärtää yksikkötestauksen suorittaminen.

5.3 Visual Studio 2010

Visual Studio on Microsoftin tekemä, ohjelmistokehittäjille tarkoitettu ohjelmankehitysympäristö. Visual Studiolla voidaan kehittää muun muassa konsolisovelluksia, työpöytäsovelluksia ja internetsovelluksia. (Avery 2005.)

Visual Studio 2010 sisältää jo valmiina työkaluja ohjelmistotestaukseen. Tässä opinnäytetyössä keskitytään vain Visual Studion yksikkötestaustyökaluun. Visual Studio 2010:ssä voidaan testitapaukset luoda manuaalisia testejä varten. Ohjelmassa voidaan myös luoda automaattisia testejä. Automaattisien testien avulla voidaan testata ohjelmaa paljon tehokkaammin, koska testitapauksia ei tarvitse suunnitella joka kerta uudestaan. NUnitiin verrattuna Visual Studion testityökalut ovat hieman käytännöllisempiä, koska ei tarvitse asentaa erikseen ohjelmia testausta varten, vaan testaustyökalut ovat valmiina. Visual Studiossa jokainen testi ajetaan omassa säikeessä, joten ei tarvitse odottaa ensimmäisen testin valmistumista, jotta voidaan siirtyä seuraavaan testiin. (MSDN 2010.)

Visual Studio 2010:llä testausluokan luominen onnistuu helposti, nopeasti ja automatisoidusti. Vain muutamalla hiiren painalluksella saadaan luotua uusi testiprojekti, joka sisältää testaukseen tarvittavat attribuutit ja metodit. Lopullinen testausluokka on kuvassa 6. Kuvasta on poistettu turhat kommentit ja jätetty vain tässä testauksessa tarvittavat tiedot näkyviin.

```
[TestClass()]
public class MatikkaLuokkaTest
{
    [TestMethod()]
    public void PlussaaNumerotTest()
    {
        MatikkaLuokka target = new MatikkaLuokka(); // TODO: Initialize to an appropriate value
        int a = 0; // TODO: Initialize to an appropriate value
        int b = 0; // TODO: Initialize to an appropriate value
        int expected = 0; // TODO: Initialize to an appropriate value
        int actual;
        actual = target.PlussaaNumerot(a, b);
        Assert.AreEqual(expected, actual);
        Assert.Inconclusive("Verify the correctness of this test method.");
    }


    [TestMethod()]
    public void JaaNumerotTest()...


    [TestMethod()]
    public void KerroNumerotTest()...


    [TestMethod()]
    public void MiinustaNumerotTest()...
}
```

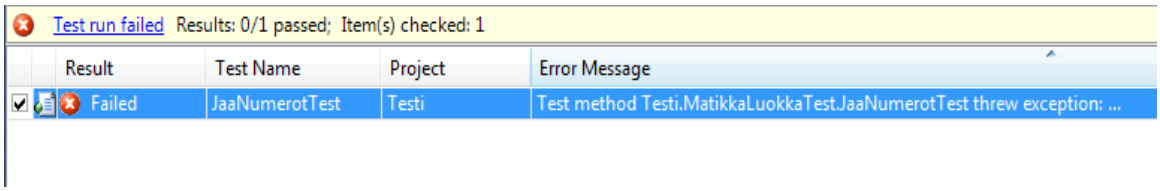
Kuva 6. Visual Studio 2010:llä tehty testausluokka.

Ajetaan esimerkkitesti JaaNumerotTest-metodin avulla. Ohjelmoinnissa nollalla jakamisella on hieman suurempi merkitys verrattuna muihin matemaattisiin operaatioihin. Jos nollalla jakamista ei ole käsitelty mitenkään ohjelmassa, jako aiheuttaa ohjelman siirtymisen poikkeustilaan tai ohjelman keskeytymiseen. Jos ohjelma rikkoontuu nollalla jaon vuoksi, on se täysin ohjelmoijan vika. Jokaisen ohjelmoijan pitäisi huolehtia, ettei tätä pääsisi tapahtumaan.

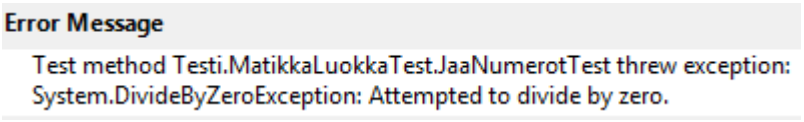
```
[TestMethod()]
public void JaaNumerotTest()
{
    MatikkaLuokka target = new MatikkaLuokka();
    int a = 5;
    int b = 0;
    int odotettu = 0;
    int oikea;
    oikea = target.JaaNumerot(a, b);
    Assert.AreEqual(odotettu, oikea);
}
```

Kuva 7. Testimetodin muutokset.

JaaNumerotTest-metodia muutettiin hieman. JaaNumerot-funtiolle annetaan arvot viisi ja nolla, eli yritetään suorittaa nollalla jako. Assert.AreEqual-metodille annetaan arvo, jonka pitäisi tulla funktiosta ja itse funktion laskema tulos.



Kuva 8. Testiajon tulokset.



Kuva 9. Testiajon tarkemmat tiedot.

Testin ajamisen jälkeen aukeaa uusi ikkuna, joka näyttää testin tulokset (kuva 8). Ikkunassa näkyvät testaukseen valitut metodit ja niiden tulokset. Koska yritettiin jakaa nollalla, testi huomauttaa virheestä. Tarkemmat tiedot virheestä näkyvät kuvassa 9. Tässä tapauksessa virheilmoitus on hyvinkin yksinkertainen, eli ohjelma meni poikkeustilaan ja ilmoitti virheestä. Nyt kun tiedetään mistä virhe johtui ja missä virhe sijaitsee, voidaan korjata asia.

```
public int JaaNumerot(int a, int b)
{
    if (a == 0 || b == 0)
        return 0;
    else
        return a / b;
}
```

Kuva 10. Funktion korjaus.

Tässä esimerkissä virhe voidaan korjata tarkastelemalla JaaNumerot-funktiolle annettuja arvoja. Nollalla jakaminen voidaan esimerkiksi korjata sillä tavoin, että jos muuttujan a tai b arvo on nolla, niin palautetaan nolla. Jos molemmat arvot ovat suurempia kuin nolla, niin palautetaan jakolaskun tulos.

5.4 NUnit

NUnit on yksikkötestausrajapinta kaikille Microsoft .NET -ohjelmointikielille. NUnit on ohjelmoitu kokonaan C#-ohjelmointikielellä. NUnit myös automatisoi yksikkötestauksen ja samalla vähentää ohjelmistokehittäjän taakkaa, ettei ohjelmoijan tarvitse koko ajan testata ohjelmakoodia suunnitellessaan tai kirjoittaessaan. (Hamilton 2004, 12-13.)

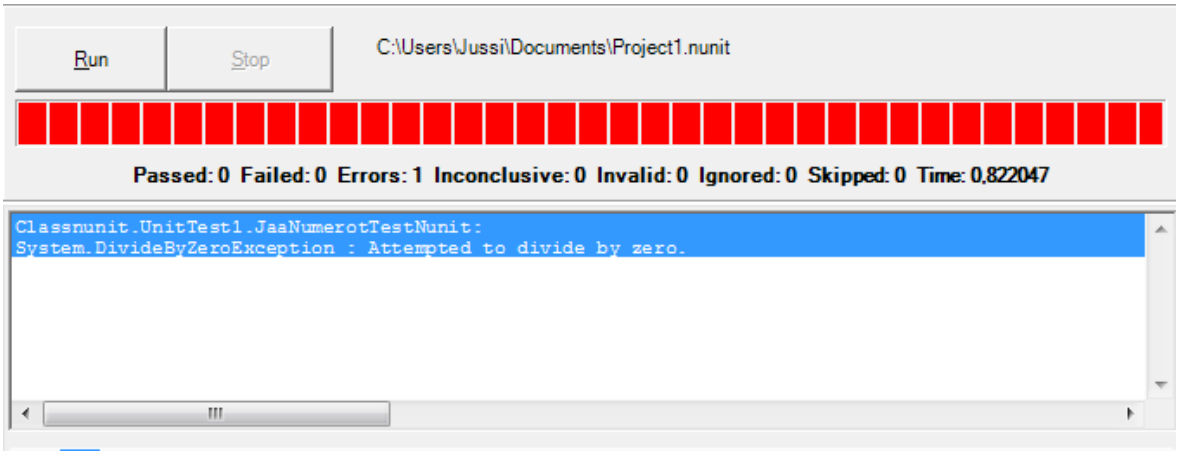
NUnitin huonoja puolia ovat muun muassa: erillinen asennuspaketti, Visual Studion kanssa ei ole suoraa integraatiota, testitapaukset täytyy luoda manuaalisesti, testitapaukset täytyy suorittaa erillisellä ohjelmalla ja testit suoritetaan yhdessä säikeessä (Arveti 2008).

NUnit täytyy ensin asentaa erikseen, jotta sitä voidaan käyttää. Asennuksen mukana tulevat tarvittavat luokkakirjastot ja erillinen ohjelma, jolla suoritetaan itse testi. NUnitin kanssa täytyy ohjelmoijan kirjoittaa testaukseen liittyvää ohjelmakoodia hieman enemmän verrattuna Visual Studion yksikkötestaustyökaluun.

```
[TestFixture]
public class UnitTest1
{
    [Test]
    public void JaaNumerotTestNunit()
    {
        MatikkaLuokka target = new MatikkaLuokka();
        int a = 5;
        int b = 0;
        int odotettu = 0;
        int oikea;
        oikea = target.JaaNumerot(a, b);
        Assert.AreEqual(odotettu, oikea);
    }
}
```

Kuva 11. NUnitia varten tehty testausluokka.

Kuvassa 11 on lisätty TestFixture- ja Test-attribuutit. Samoin on ohjelmoijan pitänyt kirjoittaa JaaNumerotTestNunit-metodi. Tässä vaiheessa ei ole paljoakaan eroa verrattuna Visual Studioon. Visual Studiossa on yksikkötestauksen luonti tehty käyttäjälle hieman yksinkertaisemmaksi.



Kuva 12. NUnitilla ajettu testi.

Jotta testi voidaan suorittaa, on käynnistettävä NUnitin oma ohjelma testausta varten. Ohjelma on hyvin yksinkertainen. Testiluokasta muodostuva luokkakirjasto on tuotava ohjelmaan, jotta testi voidaan suorittaa. Suorituksen jälkeen ohjelma ilmoittaa testauksen tulokset. Tässä tapauksessa ohjelma antaa saman virheilmoi-

tuksen, minkä Visual Studio antoi. Virhe voidaan korjata samalla tavalla kuin kuvassa 10. Kun testi ajetaan uudelleen, ei virheitä löydy.

6 TULOKSET

Työn tulokset vastasivat työn tavoitteita. Molempien työkalujen asennus ja käyttöönotto eivät vaatineet paljoa taitoa, eivätkä aikaa.

Työkalujen paremmuudesta voidaan olla montaa mieltä. Jotkut henkilöt pitävät enemmän ohjelmista, jotka ovat avointa lähdekoodia, ja toiset taas pitävät enemmän suljetuista ohjelmista. Yksikkötestauksen toteuttamisen kannalta ei ole eroa, kumpaa työkalua käyttää. Visual Studio 2010:ssä on se etu, että ohjelma on tuttu kaikille ohjelmoijille yrityksessämme ja sen vuoksi yksikkötestauksen mahdollinen kokeilu voisi olla luontevampaa Visual Studion avulla. Molemmilla työkaluilla saa saman tuloksen aikaan, joten ylivoimaista voittajaa ei ollut.

Ohjelmointia ja testausta enemmän aikaa vie testitapauksien suunnittelu. Yksikkötestausta voitaisiin kokeilla osittain tulevissa, hieman suuremmissa projekteissa. Mahdollinen kokeilu riippuu myös siitä, onko täysimittaiselle yksikkötestauksen toteuttamiselle tarpeeksi aikaa. Oma tieto ja kokemus ohjelmisto- ja yksikkötestauksesta kasvoi aika paljon ja suurella todennäköisyydellä tästä työstä voi olla myös hyötyä tulevaisuudessa.

7 YHTEENVETO

Tässä opinnäytetyössä tutkittiin yksikkötestauksen mahdollista toteuttamista. Aluksi käytiin läpi teoriaa ohjelmistotestauksesta, ohjelmistotestauksen strategioista ja ohjelmistotestauksen eri tasoista. Lopuksi tutkittiin yksikkötestausta pienen esimerkin ja kahden eri työkalun avulla. Kuten aiemmin on jo mainittu, työkalujen käyttöönotto ei vaadi paljoakaan osaamista ja onnistuu myös aloittelijoilta. Vaikka NUnitissa testaus suoritettiin erillisellä ohjelmalla, oli ohjelma yksinkertainen ja helppokäyttöinen.

Vastoinkäymisiä ei tässä työssä ollut ja kaikki testaukset sujuivat ongelmitta. Työ oli mielenkiintoinen ja riittävän haastavakin. Ennen tätä työtä, ei kirjoittajalla ollut oikeastaan yhtään kokemusta tai tietoa ohjelmisto- tai yksikkötestauksesta, mutta työn edetessä on opittu paljon uusia ja hyödyllisiä asioita.

LÄHTEET

Arveti, S. 2008. Introduction to Unit Testing Framework of VS 2008. [www-dokumentti]. Mindcracker. [Viitattu 2.3.2011]. Saatavissa: http://www.c-sharpcorner.com/UploadFile/satisharveti/ VS2008UnitTesting08132008055534AM/ VS2008UnitTesting.aspx

Avery, J. 2005. What Is Visual Studio. [www-dokumentti]. O'Reilly Media, Inc. [Viitattu 7.3.2011]. Saatavissa: http://oreilly.com/pub/a/windows/2005/08/22/whatisVisualStudio.html

Ghahrai, A. 2008. V Model. [www-dokumentti]. TestingExcellence.com. [Viitattu 19.1.2011]. Saatavissa: http://www.testingexcellence.com/v-model/

Bentley, J. E. 2005. Software Testing Fundamentals—Concepts, Roles, and Terminology. [www-dokumentti]. SAS Institute Inc. [Viitattu 20.1.2011]. Saatavissa: http://www2.sas.com/proceedings/sugi30/141-30.pdf

Davis, R. 2000. What is gray/grey box testing. [www-dokumentti]. Rob Davis. [Viitattu 28.2.2011]. Saatavissa: http://www.robdavispe.com/free2/software-qa-testing-test-tester-2210.html

Drayton, P. 2002. Introducing C# and the .NET Framework, Part 1. [www-dokumentti]. O'Reilly Media, Inc. [Viitattu 20.4.2011]. Saatavissa: http://ondotnet.com/pub/a/dotnet/excerpt/csharpnut_1/index1.html

Hamilton, B. 2004. NUnit Pocket Reference. O'Reilly Media, Inc. Gravenstein Highway North, Sebastopol. [Viitattu 5.3.2011].

Hatton, L. 1999. The Ariane 5 bug and a few lessons. [www-dokumentti]. Oakwood Computing, U.K. and the Computing Laboratory, University of Kent. [Viitattu 12.2.2011]. Saatavissa: http://leshatton.org/Documents/Ariane5_STQE499.pdf

Holkko, T. 2011. Markkinointijohtaja. Finncomm Oy. Haastattelu 6.4.2011.

Koundinya. 2010. Black Box Testing, Its Advantages and Disadvantages. [www-dokumentti]. CodeProject. [Viitattu 14.2.2011]. Saatavissa: http://www.codeproject.com/KB/testing/Black_Box_Testing.aspx

Loveland, S. & Miller, G. & Prewitt, R. & Shannon, M. 2005. Software Testing Techniques: Finding the Defects that Matter. Charles River Media, Inc. Hingham, Massachusetts. [Viitattu 1.3.2011].

Mathur, S. & Malik, S. 2010. Advancements in the V-Model. [www-dokumentti]. International Journal of Computer Applications. [Viitattu 25.1.2011]. Saatavissa: http://www.ijcaonline.org/journal/number12/pxc387425.pdf

Meerts, J. 2010. The History of Software Testing. [www-dokumentti]. Testing References. [Viitattu 14.2.2011]. Saatavissa: http://www.testingreferences.com/testinghistory.php

MSDN. 2003a. Unit Tests Overview. [www-dokumentti]. Microsoft. [Viitattu 17.1.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms182516(v=vs.80).aspx

MSDN. 2003b. Integration Testing. [www-dokumentti]. Microsoft. [Viitattu 19.1.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/aa292128(v=vs.71).aspx

MSDN. 2010. Creating and Managing Tests. [www-dokumentti]. Microsoft. [Viitattu 2.2.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/dd286729.aspx

Myers, G. J. 2004. The Art of Software Testing, Second Edition. John Wiley & Sons, Inc. Hoboken, New Jersey. [Viitattu 22.2.2011].

LKP Consulting Group. 2006. The Real Cost of Software Defects. [www-dokumentti]. LKP Consulting Group. [Viitattu 1.3.2011]. Saatavissa: http://www.lkpgroup.com/Cost%20of%20Software%20Defects.pdf

Parekh, N. 2005. Software Verification & Validation Model - An Introduction. [www-dokumentti]. Buzzle.com [Viitattu 25.2.2011]. Saatavissa: http://www.buzzle.com/editorials/4-5-2005-68117.asp

Parekh, N. 2006. Software Testing - White Box Testing Strategy. [www-dokumentti]. Buzzle.com [Viitattu 20.2.2011]. Saatavissa: http://www.buzzle.com/editorials/4-10-2005-68350.asp

Parekh, N. 2007. Software Testing - Black Box Testing Strategy. [www-dokumentti]. Buzzle.com [Viitattu 24.2.2011]. Saatavissa: http://www.buzzle.com/editorials/4-10-2005-68349

Räsänen, S. 2007. Olio-ohjelmointi. [www-dokumentti]. Savonia [Viitattu 20.4.2011]. Saatavissa: http://webd.savonia.fi/home/ktrasse/mat_tiedostot/thy8s/thy23/ohjelmointi/olio.php